CS610 Assignment 2
Name: Manish (190477)

SYSTEM CONFIGURATION:

GCC Compiler Version:gcc (Ubuntu 11.2.0-19ubuntu1) 11.2.0
Please check all the system info in file named systeminfo.

SOLUTION 1.

Consider the following code:

```
int i, j, k, l;
for (i = 0; i < 1024; i++) {
 for (j = 1; j < 1024; j++) {
  for (k = j; k < i; k++) {
   for (l = 1; l < k; l++) {
    S(i, j, k, l);
   }
  }
 }
}
```

The data dependencies for the loop are given to be (1,0,-1,1), (1,-1,0,1), (0,0,1,-1), and (0,1,0,-1). Constructing the Direction Matrix out the given Distance vectors:

$$DM = \begin{bmatrix} 1 & 0 & -1 & 1 \\ 1 & -1 & 0 & 1 \\ 0 & 0 & 1 & -1 \\ 0 & 1 & 0 & -1 \end{bmatrix}$$

1. Loop i,j,k are carrying dependencies. Loop i is carrying dependence for distance vector(1, 0, -1, 1) and (1, -1, 0, 1). Loop j is carrying dependence for distance vector(0, 1, 0, -1). Loop k is carrying dependence for distance vector(0, 0, 1, -1). Only loop l is parallel as it doesn't carry any dependence.

2. Only Valid Permutations are ijkl(which is given) and ikjl. In the rest of the 10 permutations, there is always a '-' direction as the leftmost non '0' direction in the direction vector of given 4 dependencies.

3. Sufficient conditions for loop unrolling and jamming can be obtained by observing that the complete unroll/jam of a loop is equivalent to a loop permutation that moves that loop innermost, without changing the order of other loops.
Unrolling i will give:

$$DM = \begin{bmatrix} 0 & -1 & 1 & 1 \\ -1 & 0 & 1 & 1 \\ 0 & 1 & -1 & 0 \\ 1 & 0 & -1 & 0 \end{bmatrix}$$

We can see first and the second row gives invalid Distance vectors. Hence We cannot unroll and jam Loop i.
Unrolling j will give:

$$DM = \begin{bmatrix} 1 & -1 & 1 & 0 \\ 1 & 0 & 1 & -1 \\ 0 & 1 & -1 & 0 \\ 0 & 0 & -1 & 1 \end{bmatrix}$$

We can see the fourth row gives invalid Distance vectors. Hence We cannot unroll and jam Loop j.
Unrolling k will give:

$$DM = \begin{bmatrix} 1 & 0 & 1 & -1 \\ 1 & -1 & 1 & 0 \\ 0 & 0 & -1 & 1 \\ 0 & 1 & -1 & 0 \end{bmatrix}$$

We can see the Third row gives invalid Distance vectors. Hence We cannot unroll and jam Loop k.
Unrolling l will give:

$$DM = \begin{bmatrix} 1 & 0 & -1 & 1 \\ 1 & -1 & 0 & 1 \\ 0 & 0 & 1 & -1 \\ 0 & 1 & 0 & -1 \end{bmatrix}$$

We can unroll and jam Loop l.

4. Only 2D tiling of jk is valid. As only we only valid permutations are ijkl and ikjl. Here we can only switch j and k positions.

5. jilk form of the loop:

```
for(j = 1 ; j < 1024 ; j++){
    for(i = 0; i < 1024 ; i++){
        for(l = 1; l < max((i-j)/2+1,0) ; l++) {
            for ( k = j; k < i ; k++) {
                s(i,j,k,l);
            }
        }
    }
}
```

SOLUTION 2.

This loop is performing poor because we have two statements together in which the same array is accessed in row-major by one statement and column-major by another statement. This leads to poor performance. We also don't have much content in this loop. So Operation level parallelism may not be exploited. So to solve these issues We can apply Loop permutation, Loop Unrolling and Loop fission to gain some performance in a single core.
Set of Transformation I applied with performance analysis:

```
#define N (1 << 13)

double x[N], y[N], z[N], A[N][N];

for (int i = 0; i < N; i++) {
 for (int j = 0; j < N; j++) {
  y[j] = y[j] + A[i][j]*x[i];
  z[j] = z[j] + A[j][i]*x[i];
 }
}
```

1.

Transformation	Flag	Performance Analysis(No of times wrt. reference)
Loop Permutation (i,j) is changed to (j,i)	-O2	0.96(Almost same performance)
Loop Permutation (i,j) is changed to (j,i)	-O3	0.54(Slow Down)

We didn't get any benefit from loop permutation because the first statement in the loop is exploiting spatial locality in the reference version but after permutation S2 is exploiting spatial locality but on S1 we are losing out.

2.

Transformation	Flag	Performance Analysis(No of times wrt. reference)
Loop Fission	-O2	0.83(Slow Down)
Loop Fission	-O3	0.82(Slow Down)

We didn't get performance benefit because we are reducing the statement in the loop which reduces operation level parallelism and variable reuse like x[i].

3.

Transformation	Flag	Performance Analysis(No of times wrt. reference)
Loop Unrolling 4	-O2	1.1(Speed Up)
Loop Unrolling 4	-O3	0.8(Slow Down)
Loop Unrolling 8	-O2	0.90(Slow Down)
Loop Unrolling 8	-O3	0.39(Slow Down)
Loop Unrolling 16	-O2	0.85(Slow Down)
Loop Unrolling 16	-O3	0.37(Slow Down)
Loop Unrolling 32	-O2	0.78(Slow Down)
Loop Unrolling 32	-O3	0.36(Slow Down)

We didn't get much benefits in more loop rolling may be because of increase in register usage spills data back to memory which may result in a slow down.

4.

Transformation	Flag	Performance Analysis(No of times wrt. reference)
Loop Fission and Loop permutation of 2nd loop	-O2	2.75(Speed Up)
Loop Fission and Loop permutation of 2nd loop	-O3	2.21(Speed Up)

This speed that we are getting is expected because we have 2 statements, one having row-major access and the other having column major access. If we split the loop and permute the 2nd loop

then S2 also becomes row-major access which results in speed up.

5.

Transformation	Flag	Performance Analysis(No of times wrt. referen
Loop Fission + Loop permutation + Loop Unrolling 4	-O2	2.99(Speed Up)
Loop Fission + Loop permutation + Loop Unrolling 4	-O2	2.3(Speed Up)
Loop Fission + Loop permutation + Loop Unrolling 8	-O2	2.95(Speed Up)
Loop Fission + Loop permutation + Loop Unrolling 8	-O2	2.0(Speed Up)

This speed is closer to the loop fission + permutation. Loop rolling helps a little bit. Loop rolled by 4 is giving us good performance as we are not going out of register.

6.

Transformation	Flag	Performance Analysis(No of times wrt. reference)
AVX intrinsics	-O2	2.3(Speed Up)
AVX intrinsics	-O2	1.7(Speed Up)

SOLUTION 3.

```
#define N (1 << 12)
#define ITER 100

double A[N][N];

for (int k=0; k<ITER; k++) {
 for (int i=1; i<N; i++) {
  for (int j=0; j<N-1; j++) {
   A[i][j+1] = A[i-1][j+1] + A[i][j+1];
  }
 }
}
```

The distance vector of the dependency in the loop is (*,1,0). We cannot permute k to permutations like ikj, jik and ijk as if k is ”-” then this gives us invalid dependencies for the permutation mentioned. Only Valid permutations are kij(given), kji, jki.
Different Transformations we can Try:
1. Loop permutation (Valid permutations are kji, jki and kij) and then parallelize the j loop.
2. Loop Unrolling 8/16
Results of these Transformations:
1. Loop Permutation + running parallel j loop:
kij(given): 0.92(Slow)
It may be because we can only parallelize the j loop. In which we are not doing anything significant. We always want to parallelize the outer loops such that threads’ overheads can be compensated.
kji: 0.13(Very slow)
This loop is slow as we expect because we changed the row-major access to column-major access.
jki: 0.25(Very slow)

This loop is slow as we expect because we changed the row-major access to column-major access. But this is faster than kji because we are more work in each iteration as the j loop is outermost which we parallelized.

2. Loop Unrolling + Permutation + running j loop parallel:
kij + Loop Unrolled 8: 0.96(Almost same)
Loop Unrolling + permutation helped over just loop permutation.
jki + Loop Unrolled 8: 0.17(Very slow)
Loop Unrolling gives poor performance than just loop permutation.

SOLUTION 4.

1. OPENMP parallel for without reduction: 1.28 (Speed Up)
2. OPENMP parallel using reduction: 1.39 (Speed Up)
3. OPENMP Tasks: 0.73(Slow down)

SOLUTION 5.

1. OpenMP: 0.41 (Slow Down)
2. SSE: 1.23 (Speed Up)
3. AVX: 1.31 (Speed Up)